Python Collections Module - Comprehensive Tutorial

The `collections` module provides specialized container datatypes as alternatives to Python's built-in containers like `dict`, `list`, `set`, and `tuple`. These specialized containers offer additional functionality and performance optimizations for specific use cases.

Table of Contents

Counter

Purpose: Count hashable objects. It's a subclass of dict for counting.

When to use:

- Counting occurrences of elements
- Finding most/least common items
- Mathematical operations on counts

Basic Usage

```python
from collections import Counter

# Creating a Counter
text = "hello world"
counter = Counter(text)
print(counter)  # Counter({'l': 3, 'o': 2, 'h': 1, 'e': 1, ' ': 1, 'w': 1, 'r': 1, 'd': 1})

# From a list
colors = ['red', 'blue', 'red', 'green', 'blue', 'blue']
color_count = Counter(colors)
print(color_count)  # Counter({'blue': 3, 'red': 2, 'green': 1})

# From a dictionary
counter_dict = Counter({'a': 3, 'b': 1, 'c': 2})
print(counter_dict)  # Counter({'a': 3, 'c': 2, 'b': 1})
```

Useful Methods

```python
from collections import Counter

votes = Counter(['alice', 'bob', 'alice', 'charlie', 'bob', 'alice'])

# Most common elements
print(votes.most_common())     # [('alice', 3), ('bob', 2), ('charlie', 1)]
print(votes.most_common(2))    # [('alice', 3), ('bob', 2)]

# Total count
print(votes.total())          # 6

# Elements (iterator over elements)
print(list(votes.elements()))  # ['alice', 'alice', 'alice', 'bob', 'bob', 'charlie']

# Update with more data
votes.update(['alice', 'david'])
print(votes)  # Counter({'alice': 4, 'bob': 2, 'charlie': 1, 'david': 1})

# Subtract counts
votes.subtract(['alice', 'bob'])
print(votes)  # Counter({'alice': 3, 'bob': 1, 'charlie': 1, 'david': 1})
```

Mathematical Operations

```python
from collections import Counter

c1 = Counter(['a', 'b', 'c', 'a', 'b'])
c2 = Counter(['a', 'b', 'b', 'd'])

# Addition
print(c1 + c2)  # Counter({'a': 3, 'b': 4, 'c': 1, 'd': 1})

# Subtraction
print(c1 - c2)  # Counter({'c': 1, 'a': 1})

# Intersection (minimum)
print(c1 & c2)  # Counter({'a': 1, 'b': 1})

# Union (maximum)
print(c1 | c2)  # Counter({'a': 2, 'b': 2, 'c': 1, 'd': 1})
```

Practical Example: Word Frequency Analysis

```python
from collections import Counter
import re

def analyze_text(text):
    # Clean and split text
    words = re.findall(r'\b\w+\b', text.lower())
    word_count = Counter(words)

    print(f"Total words: {word_count.total()}")
    print(f"Unique words: {len(word_count)}")
    print(f"Most common words: {word_count.most_common(5)}")

    return word_count

text = "Python is great. Python is powerful. Programming in Python is fun."
analyze_text(text)
```

---

defaultdict

Purpose: Dictionary that provides a default value for missing keys.

When to use:

- Avoiding KeyError exceptions
- Grouping items by key
- Creating nested data structures

Basic Usage

```python
from collections import defaultdict

# Regular dict would raise KeyError
regular_dict = {}
try:
    print(regular_dict['missing_key'])
except KeyError as e:
    print(f"KeyError: {e}")

# defaultdict provides default value
dd = defaultdict(int)  # default value is 0
print(dd['missing_key'])  # 0
dd['existing_key'] = 5
print(dd['existing_key'])  # 5
```

Common Default Factory Functions

```python
from collections import defaultdict

# int: default value 0
int_dd = defaultdict(int)
int_dd['count'] += 1
print(int_dd)  # defaultdict(<class 'int'>, {'count': 1})

# list: default value []
list_dd = defaultdict(list)
list_dd['items'].append('first')
list_dd['items'].append('second')
print(list_dd)  # defaultdict(<class 'list'>, {'items': ['first', 'second']})

# set: default value set()
set_dd = defaultdict(set)
set_dd['tags'].add('python')
set_dd['tags'].add('tutorial')
print(set_dd)  # defaultdict(<class 'set'>, {'tags': {'python', 'tutorial'}})

# Custom default factory
def default_value():
    return "N/A"

custom_dd = defaultdict(default_value)
print(custom_dd['unknown'])  # N/A
```

Practical Example: Grouping Data

python

```python
from collections import defaultdict

# Group students by grade
students = [
    ('Alice', 'A'),
    ('Bob', 'B'),
    ('Charlie', 'A'),
    ('David', 'C'),
    ('Eve', 'B')
]

# Using defaultdict
grade_groups = defaultdict(list)
for name, grade in students:
    grade_groups[grade].append(name)

print(dict(grade_groups))
# {'A': ['Alice', 'Charlie'], 'B': ['Bob', 'Eve'], 'C': ['David']}

# Creating nested defaultdict
nested_dd = defaultdict(lambda: defaultdict(int))
nested_dd['fruits']['apple'] = 5
nested_dd['fruits']['banana'] = 3
nested_dd['vegetables']['carrot'] = 2

print(dict(nested_dd))
# {'fruits': defaultdict(<class 'int'>, {'apple': 5, 'banana': 3}),
#  'vegetables': defaultdict(<class 'int'>, {'carrot': 2})}
```

OrderedDict

Purpose: Dictionary that maintains insertion order of keys.

When to use:

- When order matters (though regular dict maintains order in Python 3.7+)
- Moving items to end/beginning
- LRU cache implementation

Basic Usage

```python
from collections import OrderedDict

# Regular dict (maintains order in Python 3.7+)
regular_dict = {'a': 1, 'b': 2, 'c': 3}
print(regular_dict)  # {'a': 1, 'b': 2, 'c': 3}

# OrderedDict
od = OrderedDict()
od['first'] = 1
od['second'] = 2
od['third'] = 3
print(od)  # OrderedDict([('first', 1), ('second', 2), ('third', 3)])
```

Unique Methods

```python
from collections import OrderedDict

od = OrderedDict([('a', 1), ('b', 2), ('c', 3)])

# Move to end
od.move_to_end('a')
print(od)  # OrderedDict([('b', 2), ('c', 3), ('a', 1)])

# Move to beginning
od.move_to_end('c', last=False)
print(od)  # OrderedDict([('c', 3), ('b', 2), ('a', 1)])

# Pop last item
last_item = od.popitem(last=True)
print(f"Popped: {last_item}")  # Popped: ('a', 1)
print(od)  # OrderedDict([('c', 3), ('b', 2)])

# Pop first item
first_item = od.popitem(last=False)
print(f"Popped: {first_item}")  # Popped: ('c', 3)
print(od)  # OrderedDict([('b', 2)])
```

Practical Example: LRU Cache Implementation

```python
from collections import OrderedDict

class LRUCache:
    def __init__(self, capacity):
        self.capacity = capacity
        self.cache = OrderedDict()

    def get(self, key):
        if key in self.cache:
            # Move to end (mark as recently used)
            self.cache.move_to_end(key)
            return self.cache[key]
        return None

    def put(self, key, value):
        if key in self.cache:
            # Update existing key
            self.cache[key] = value
            self.cache.move_to_end(key)
        else:
            # Add new key
            if len(self.cache) >= self.capacity:
                # Remove least recently used item
                self.cache.popitem(last=False)
            self.cache[key] = value

    def __str__(self):
        return str(dict(self.cache))

# Usage
lru = LRUCache(3)
lru.put('a', 1)
lru.put('b', 2)
lru.put('c', 3)
print(lru)  # {'a': 1, 'b': 2, 'c': 3}

lru.get('a')  # Access 'a'
lru.put('d', 4)  # This will evict 'b' (least recently used)
print(lru)  # {'c': 3, 'a': 1, 'd': 4}
```

---

deque

Purpose: Double-ended queue optimized for adding/removing elements from both ends.

When to use:

- Need efficient append/pop operations from both ends
- Implementing queues, stacks, or circular buffers
- Sliding window operations

Basic Usage

```python
from collections import deque

# Creating deque
dq = deque(['a', 'b', 'c'])
print(dq)  # deque(['a', 'b', 'c'])

# From iterable with maximum length
dq_limited = deque([1, 2, 3], maxlen=3)
print(dq_limited)  # deque([1, 2, 3], maxlen=3)
```

Operations

python

```python
from collections import deque

dq = deque(['b', 'c', 'd'])

# Add to right (end)
dq.append('e')
print(dq)  # deque(['b', 'c', 'd', 'e'])

# Add to left (beginning)
dq.appendleft('a')
print(dq)  # deque(['a', 'b', 'c', 'd', 'e'])

# Remove from right
right_item = dq.pop()
print(f"Popped from right: {right_item}")  # e
print(dq)  # deque(['a', 'b', 'c', 'd'])

# Remove from left
left_item = dq.popleft()
print(f"Popped from left: {left_item}")  # a
print(dq)  # deque(['b', 'c', 'd'])

# Extend from both sides
dq.extend(['e', 'f'])
dq.extendleft(['z', 'y'])  # Note: items are added one by one from left
print(dq)  # deque(['y', 'z', 'b', 'c', 'd', 'e', 'f'])

# Rotate
dq.rotate(2)  # Rotate right by 2
print(dq)  # deque(['e', 'f', 'y', 'z', 'b', 'c', 'd'])

dq.rotate(-3)  # Rotate left by 3
print(dq)  # deque(['z', 'b', 'c', 'd', 'e', 'f', 'y'])
```

Practical Example: Sliding Window Maximum

python

```python
from collections import deque

def sliding_window_maximum(nums, k):
    """Find maximum in each sliding window of size k"""
    if not nums or k == 0:
        return []

    # Deque stores indices
    dq = deque()
    result = []

    for i in range(len(nums)):
        # Remove indices outside current window
        while dq and dq[0] <= i - k:
            dq.popleft()

        # Remove indices with smaller values
        while dq and nums[dq[-1]] < nums[i]:
            dq.pop()

        dq.append(i)

        # Add maximum to result when window is complete
        if i >= k - 1:
            result.append(nums[dq[0]])

    return result

# Usage
nums = [1, 3, -1, -3, 5, 3, 6, 7]
k = 3
result = sliding_window_maximum(nums, k)
print(result)  # [3, 3, 5, 5, 6, 7]
```

Practical Example: Circular Buffer

python

```python
from collections import deque

class CircularBuffer:
    def __init__(self, size):
        self.buffer = deque(maxlen=size)
        self.size = size

    def add(self, item):
        self.buffer.append(item)

    def get_all(self):
        return list(self.buffer)

    def is_full(self):
        return len(self.buffer) == self.size

# Usage
buffer = CircularBuffer(3)
for i in range(5):
    buffer.add(i)
    print(f"Added {i}: {buffer.get_all()}")

# Output:
# Added 0: [0]
# Added 1: [0, 1]
# Added 2: [0, 1, 2]
# Added 3: [1, 2, 3]  # 0 was removed
# Added 4: [2, 3, 4]  # 1 was removed
```

---

namedtuple

Purpose: Create tuple subclasses with named fields.

When to use:

- Need lightweight, immutable data structures
- Want to access tuple elements by name
- Alternative to classes for simple data containers

Basic Usage

python

```python
from collections import namedtuple

# Define a namedtuple
Point = namedtuple('Point', ['x', 'y'])
p1 = Point(10, 20)
print(p1)  # Point(x=10, y=20)
print(p1.x, p1.y)  # 10 20

# Different ways to define fields
Person = namedtuple('Person', 'name age city')  # Space-separated
Employee = namedtuple('Employee', ['name', 'id', 'department'])  # List
```

Methods and Properties

python

```python
from collections import namedtuple

Point = namedtuple('Point', ['x', 'y'])
p = Point(3, 4)

# Access by index (like regular tuple)
print(p[0], p[1])  # 3 4

# Access by name
print(p.x, p.y)  # 3 4

# Convert to dict
print(p._asdict())  # {'x': 3, 'y': 4}

# Replace values (returns new instance)
p2 = p._replace(x=5)
print(p2)  # Point(x=5, y=4)

# Get field names
print(Point._fields)  # ('x', 'y')

# Create from iterable
coords = [7, 8]
p3 = Point._make(coords)
print(p3)  # Point(x=7, y=8)
```

Practical Example: Student Record System

```python
from collections import namedtuple

# Define student record
Student = namedtuple('Student', ['name', 'age', 'grade', 'gpa'])

# Create students
students = [
    Student('Alice', 20, 'A', 3.8),
    Student('Bob', 19, 'B', 3.2),
    Student('Charlie', 21, 'A', 3.9),
    Student('David', 20, 'C', 2.8)
]

# Find students with high GPA
high_gpa_students = [s for s in students if s.gpa >= 3.5]
print("High GPA students:")
for student in high_gpa_students:
    print(f"  {student.name}: {student.gpa}")

# Calculate average GPA by grade
from collections import defaultdict
grade_gpa = defaultdict(list)
for student in students:
    grade_gpa[student.grade].append(student.gpa)

for grade, gpas in grade_gpa.items():
    avg_gpa = sum(gpas) / len(gpas)
    print(f"Grade {grade} average GPA: {avg_gpa:.2f}")
```

Extending namedtuple

```python
from collections import namedtuple

# Create a namedtuple with methods
Point = namedtuple('Point', ['x', 'y'])

class Point2D(Point):
    def distance_from_origin(self):
        return (self.x ** 2 + self.y ** 2) ** 0.5

    def distance_from(self, other):
        return ((self.x - other.x) ** 2 + (self.y - other.y) ** 2) ** 0.5

    def __str__(self):
        return f"Point2D({self.x}, {self.y})"

# Usage
p1 = Point2D(3, 4)
p2 = Point2D(6, 8)
print(p1.distance_from_origin())  # 5.0
print(p1.distance_from(p2))  # 5.0
```

ChainMap

Purpose: Combine multiple mappings into a single view.

When to use:

- Merging multiple dictionaries
- Creating hierarchical configurations
- Implementing variable scoping

Basic Usage

python

```python
from collections import ChainMap

# Combine multiple dictionaries
dict1 = {'a': 1, 'b': 2}
dict2 = {'c': 3, 'd': 4}
dict3 = {'e': 5, 'f': 6}

combined = ChainMap(dict1, dict2, dict3)
print(combined)  # ChainMap({'a': 1, 'b': 2}, {'c': 3, 'd': 4}, {'e': 5, 'f': 6})
print(combined['a'])  # 1
print(combined['c'])  # 3

# Keys and values
print(list(combined.keys()))    # ['e', 'f', 'c', 'd', 'a', 'b']
print(list(combined.values()))  # [5, 6, 3, 4, 1, 2]
```

Precedence and Updates

python

```python
from collections import ChainMap

# First mapping has highest priority
dict1 = {'a': 1, 'b': 2}
dict2 = {'a': 10, 'c': 3}  # 'a' is in both dicts

cm = ChainMap(dict1, dict2)
print(cm['a'])  # 1 (from dict1, not dict2)

# Updates affect only the first mapping
cm['d'] = 4
print(dict1)  # {'a': 1, 'b': 2, 'd': 4}
print(dict2)  # {'a': 10, 'c': 3} (unchanged)

# Delete from first mapping
del cm['b']
print(dict1)  # {'a': 1, 'd': 4}
```

Methods

python

```python
from collections import ChainMap

base_config = {'host': 'localhost', 'port': 8080, 'debug': False}
user_config = {'port': 9000, 'debug': True}

config = ChainMap(user_config, base_config)

# Create new child
child_config = config.new_child({'timeout': 30})
print(child_config)
# ChainMap({'timeout': 30}, {'port': 9000, 'debug': True}, {'host': 'localhost', 'port': 8080, 'debug': False})

# Access parent
print(child_config.parents)
# ChainMap({'port': 9000, 'debug': True}, {'host': 'localhost', 'port': 8080, 'debug': False})

# Get all mappings
print(child_config.maps)
# [{'timeout': 30}, {'port': 9000, 'debug': True}, {'host': 'localhost', 'port': 8080, 'debug': False}]
```

Practical Example: Configuration Management

python

```python
from collections import ChainMap
import os

# Configuration hierarchy: command line > environment > config file > defaults
def load_config():
    # Default configuration
    defaults = {
        'host': 'localhost',
        'port': 8080,
        'debug': False,
        'database_url': 'sqlite:///app.db'
    }

    # Configuration file
    config_file = {
        'host': 'production.example.com',
        'port': 80,
        'database_url': 'postgresql://user:pass@db.example.com/app'
    }

    # Environment variables
    env_vars = {}
    if 'APP_HOST' in os.environ:
        env_vars['host'] = os.environ['APP_HOST']
    if 'APP_PORT' in os.environ:
        env_vars['port'] = int(os.environ['APP_PORT'])
    if 'APP_DEBUG' in os.environ:
        env_vars['debug'] = os.environ['APP_DEBUG'].lower() == 'true'

    # Command line arguments (simulated)
    cmd_args = {'debug': True}  # --debug flag was passed

    # Create configuration chain (highest priority first)
    config = ChainMap(cmd_args, env_vars, config_file, defaults)
    return config

# Usage
config = load_config()
print(f"Host: {config['host']}")
print(f"Port: {config['port']}")
print(f"Debug: {config['debug']}")
print(f"Database: {config['database_url']}")
```

UserDict, UserList, UserString

Purpose: Base classes for creating custom dictionary, list, and string-like objects.

When to use:

- Creating custom container classes
- Need to override specific methods
- Want to add functionality to built-in types

UserDict Example

```python
from collections import UserDict

class CaseInsensitiveDict(UserDict):
    """Dictionary that ignores case for string keys"""

    def __setitem__(self, key, value):
        if isinstance(key, str):
            key = key.lower()
        super().__setitem__(key, value)

    def __getitem__(self, key):
        if isinstance(key, str):
            key = key.lower()
        return super().__getitem__(key)

    def __contains__(self, key):
        if isinstance(key, str):
            key = key.lower()
        return super().__contains__(key)

    def __delitem__(self, key):
        if isinstance(key, str):
            key = key.lower()
        super().__delitem__(key)

# Usage
ci_dict = CaseInsensitiveDict()
ci_dict['Name'] = 'Alice'
ci_dict['AGE'] = 30

print(ci_dict['name'])  # Alice
print(ci_dict['age'])   # 30
print('NAME' in ci_dict)  # True
```

UserList Example

python

```python
from collections import UserList

class NumberList(UserList):
    """List that only accepts numbers"""

    def __init__(self, initlist=None):
        super().__init__()
        if initlist is not None:
            for item in initlist:
                self.append(item)

    def append(self, item):
        if not isinstance(item, (int, float)):
            raise TypeError(f"Only numbers allowed, got {type(item).__name__}")
        super().append(item)

    def extend(self, other):
        for item in other:
            self.append(item)  # Use our custom append method

    def sum(self):
        return sum(self.data)

    def average(self):
        return self.sum() / len(self.data) if self.data else 0

# Usage
num_list = NumberList([1, 2, 3, 4, 5])
print(num_list.sum())      # 15
print(num_list.average())  # 3.0

num_list.append(6)
print(num_list)  # [1, 2, 3, 4, 5, 6]

try:
    num_list.append('invalid')
except TypeError as e:
    print(e)  # Only numbers allowed, got str
```

UserString Example

python

```python
from collections import UserString

class ReversibleString(UserString):
    """String that can be easily reversed"""

    def reverse(self):
        return ReversibleString(self.data[::-1])

    def is_palindrome(self):
        clean = ''.join(c.lower() for c in self.data if c.isalnum())
        return clean == clean[::-1]

    def word_count(self):
        return len(self.data.split())

# Usage
text = ReversibleString("A man a plan a canal Panama")
print(text.reverse())      # amanaP lanac a nalp a nam A
print(text.is_palindrome())  # True
print(text.word_count())   # 7

# Still works like a regular string
print(text.upper())       # A MAN A PLAN A CANAL PANAMA
print(text[0:5])         # A man
```

Performance Considerations

Counter vs Regular Dict

python

```python
from collections import Counter
import time

# Large dataset
data = ['apple'] * 1000 + ['banana'] * 800 + ['orange'] * 1200

# Using Counter
start = time.time()
counter = Counter(data)
counter_time = time.time() - start

# Using regular dict
start = time.time()
regular_dict = {}
for item in data:
    regular_dict[item] = regular_dict.get(item, 0) + 1
dict_time = time.time() - start

print(f"Counter time: {counter_time:.6f}s")
print(f"Regular dict time: {dict_time:.6f}s")
```

deque vs List Performance

python

```python
from collections import deque
import time

# Compare append/pop performance
def test_performance(container, n=100000):
    start = time.time()
    for i in range(n):
        container.append(i)

    for i in range(n):
        container.pop()

    return time.time() - start

# Test with list
list_time = test_performance([])
print(f"List time: {list_time:.6f}s")

# Test with deque
deque_time = test_performance(deque())
print(f"Deque time: {deque_time:.6f}s")

# Test left operations (where deque shines)
def test_left_operations(container, n=10000):
    start = time.time()
    for i in range(n):
        if hasattr(container, 'appendleft'):
            container.appendleft(i)
        else:
            container.insert(0, i)

    for i in range(n):
        if hasattr(container, 'popleft'):
            container.popleft()
        else:
            container.pop(0)

    return time.time() - start

list_left_time = test_left_operations([])
deque_left_time = test_left_operations(deque())

print(f"List left operations: {list_left_time:.6f}s")
print(f"Deque left operations: {deque_left_time:.6f}s")
```

Summary

The `collections` module provides powerful alternatives to built-in Python containers:

- **Counter**: Perfect for counting and frequency analysis
- **defaultdict**: Eliminates KeyError and simplifies code
- **OrderedDict**: When you need guaranteed ordering (less relevant in Python 3.7+)
- **deque**: Efficient double-ended operations
- **namedtuple**: Lightweight, immutable data structures
- **ChainMap**: Combine multiple mappings with precedence
- **UserDict/UserList/UserString**: Base classes for custom containers

Choose the right tool based on your specific needs:

- Need to count things? Use `Counter`
- Grouping data? Use `defaultdict`
- Queue operations? Use `deque`
- Simple data structures? Use `namedtuple`
- Combining configurations? Use `ChainMap`
- Custom container behavior? Use `UserDict/UserList/UserString`

Each of these specialized containers can make your code more efficient, readable, and maintainable when used appropriately.